React Native Performance Evaluation

Rasmus Eskola

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 29.05.2018

Supervisor

Prof. Antti Ylä-Jääski

Advisor

MSc Tuomas Paasonen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Rasmus Eskola

Title React Native Performance Evaluation

Degree programme Computer, Communication and Information Sciences

Major Security and Cloud Computing **Code of major** SCI3084

Supervisor Prof. Antti Ylä-Jääski

Advisor MSc Tuomas Paasonen

Date 29.05.2018 **Number of pages** 54+1 **Language** English

Abstract

Smartphones have become an ubiquitous device for people, and there are multiple mobile platforms to choose from. For mobile developers this means more work, as they will need to learn the tools and technologies unique to each platform, and develop their application separately for each platform. Cross platform tools such as React Native promise a solution where developers can use the same tools and technologies across different platforms.

A React Native application is essentially a JavaScript application that controls native user interface components. As such, a React Native application has to perform more background work compared to an equivalent native mobile application. This thesis studies whether React Native carries any meaningful performance penalties with it on the Android platform, and whether it is possible to work around these problems. It aims to provide some insight into the performance figures of React Native for both developers considering the technology and researchers wanting to do further research. Methods for measuring application launch times, render latency of components, navigation latency and list scrolling are presented. In all but the last case, the measurements can be directly compared between a React Native and an Android application to find out the exact overhead of React Native in each situation.

The findings indicate that React Native does incur meaningful performance penalties compared to native code. In many cases the performance hit is not significant enough to cause user frustration, but especially on older devices common operations such as application launch and component rendering are noticeably slower and may have up to 10 times longer latency than the native equivalent. On modern devices the overhead of React Native is less noticeable, making React Native a better fit when targeting newer hardware.

Keywords React Native, Mobile, JavaScript, Cross-platform, Performance, Benchmark

Aalto University
School of Science

Aalto-universitetet, PB 11000, 00076 AALTO
www.aalto.fi
sammandrag av diplomarbetet

Författare Rasmus Eskola

Titel React Native Performance Evaluation

Utbildningsprogram Data-, informations- och kommunikations teknik

Huvudämne Security and Cloud Computing **Huvudämnets kod** SCI3084

Övervakare Prof. Antti Ylä-Jääski

Handledare DI Tuomas Paasonen

Datum 29.05.2018 **Sidantal** 54+1 **Språk** Engelska

Sammandrag

Smarttelefoner har blivit allestädes närvarande bland folk, och det finns flera olika plattformer att välja mellan. För mobilutvecklare innebär detta mera arbete, eftersom de måste behärska diverse plattformers unika verktyg samt teknologier, och dessutom utveckla sina applikationer för varje platform skilt. Plattformsoberoende teknologier som t.ex. React Native erbjuder en lösning som möjliggör användningen av samma verktyg och teknologier för flera olika plattformer.

En React Native applikation är i grund och botten en JavaScript applikation som kan rita upp nativa användargränssnittskomponenter. Detta betyder att en React Native applikation är tvungen att göra mera arbete jämfört med en nativapplikation. Detta diplomarbete undersöker om huruvida React Native har några betydelsefulla prestandaproblem på Android plattformen, och om det är möjligt att kringgå sådana problem. Arbetets syfte är att ge en insikt i React Natives prestanda åt utvecklare som överväger att använda teknologin, samt åt forskare som vill göra vidare forskning inom ämnet. Metoder presenteras för att mäta applikationers starttid, komponenters ritningstid, latens i hantering av navigation, samt rullandet av en lista. I alla utom det sista fallet kan resultaten jämföras direkt mellan React Native och Android, vilket ger oss exakt information över hur mycket extra beräkningstid React Native behöver i respektive situation.

Undersökningen påvisar att React Native är betydligt långsammare än nativkod i vissa fall. Ofta är skillnaden inte tillräckligt stor för att orsaka frustration hos användaren, men speciellt på äldre telefoner kan vanliga händelser som applikationens starttid eller komponenters rittid vara till och med 10 ggr. långsammare i en React Native applikation jämfört med en ekvivalent Android applikation. På moderna telefoner är skillnaden mellan React Native och nativkod inte lika stor, vilket gör att React Native passar bättre in ifall applikationens målgrupp använder nyare hårdvara.

Nyckelord React Native, Mobil, JavaScript, Plattformsoberoende, Prestanda

Contents

Abbreviations

Abbreviations

UI	user interface
GPU	graphics processing unit
FPS	frames per second
JS	JavaScript
API	Application Programming Interface
CLI	Command-line Interface

1 Introduction

Smartphones have become an ubiquitous device with an important place in people's everyday lives. Wherever we go, we carry our smartphones around, and they are always ready to perform day-to-day actions such as messaging, entertainment, payments and so on. Largely what sets modern smartphones apart from older feature phones is the easy ability to run third party applications or "apps", expanding the possibilities of what can be done with the device.

From the user's standpoint this is great, as there are plenty of apps available for the user's platform, and they can be sure to find an app for almost any need. However, with multiple mobile platforms to target, each with their own set of tools and programming languages, this presents a problem to the app developers. Developers need to essentially write the same app multiple times, once for each platform. With Microsoft phasing out Windows Phone [46], there are now two big smartphone platforms remaining: Android by Google and iOS by Apple [7]. This means that third party developers writing apps have to support two separate platforms if they want to reach the vast majority of smartphone users.

There have been several solutions attempting to remedy this situation, with frameworks such as Cordova and PhoneGap being driving forces behind the hybrid apps concept. They work by embedding a web browser into an app, allowing app developers to write their apps using HTML elements, CSS and JavaScript. These solutions allow a large amount of code sharing between platforms, but the downsides are poor performance and a lack of native feel in the apps [32] [49].

A more recent development provides a middle ground between hybrid apps and native apps. Instead of writing the entire application in a web browser environment using JavaScript, HTML and CSS, developers can write the application code in JavaScript and let it control native UI components instead of browser elements. React Native by Facebook is a framework that works in this manner. It allows writing application code in JavaScript, and rendering the user interface (UI) from JavaScript using native UI components of the mobile platform. The advantage compared to hybrid apps is that React Native uses true native components instead of HTML elements, and as such React Native apps will be more responsive and feel more like a native app rather than a web page [36].

React Native is based on React, which in turn is a library for rendering user interfaces in web browsers. A core idea of React is to abstract away the Document Object Model (DOM) of the web platform. Instead, React developers deal with functional units called React components. React automatically computes and performs only the necessary changes to the DOM tree based on the output of these components. [47] React Native builds on this concept, but instead of rendering HTML elements to the web browser's DOM, React Native will render native user interfaces using the same fundamental UI components as native mobile apps do.

A major advantage of applying React's component model to mobile apps is the ability to use common tools and technologies across mobile platforms, while still maintaining the native look and feel of the apps. Developers can share a majority of their codebase across platforms supported by React Native [27], and they are able to

use JavaScript and React to build their cross-platform application.

1.1 Problem statement

While React Native promises performant applications, there is still some overhead involved compared to truly native applications. The application's code runs inside of a JavaScript virtual machine which handles all the app logic, orchestrates the UI, makes simple computations, etc. After all of this has been done, the results need to be sent over to the native thread, where the UI can finally be rendered with new results. Time spent doing this work adds up, and could potentially have been done faster in native code.

Most mobile phones today run their display at a refresh rate of 60 Hz [38] , which also sets a target for how often applications should render a new frame in order to appear responsive and smooth. Even though a React Native application may be less performant than a truly native application, it won't matter appearance-wise as long as it manages to hit 60 frames per second (FPS).

Another important factor is latency, or the time it takes from user input until the device has responded in some way. In the history of the computer science field, many studies have been conducted on the impact of latency in user interfaces. It is important that latency stays as low as possible, or the user will be distracted from their task at hand and may get frustrated [42]. For React Native this means that there is not much room for additional latency as compared to the native platform, and ideally a React Native application should respond as quickly as a native application.

Naturally there are other factors to performance than the framerate and latency of an application. Other problems include battery life drain, cpu usage, memory usage and disk space requirements. In this study we will be focusing on latency, framerate and tasks that cause the framerate to drop below 60 FPS. Dropping below 60 FPS causes extra latency, which means that the user will experience a slowdown and the app will appear unresponsive [34].

Not much research exists on React Native's performance specifically yet. This study aims to shed some light on the subject by testing and comparing common UI components in mobile apps between truly native apps and React Native apps. We measure how long these components take to render as well as how long they take to react to user input both in native and React Native apps. We also compare various React Native components to see which is most suitable for a certain situation. Common components to be tested include basic text views, lists of data, interaction with basic mobile UI components such as buttons and scrollable views.

1.2 Research questions

This thesis will attempt to clarify the performance impacts behind React Native compared to native code. The aim is to provide insight into performance aspects of React Native so that developers can make more informed decisions about the framework, and to work as an aid for further React Native research.

The research questions can be summed up as follows:

- Is there a meaningful hit to performance when using React Native?
- Are there any ways to work around common performance problems in React Native?

1.3 Structure of the Thesis

This thesis contains seven chapters. The first and second chapters introduce the subject, as well as discuss existing related work. Chapter three explains the methods used for collecting data. Chapter four goes into the implementation details of the used methods. Chapters five and six evaluate and discuss the results, and chapter seven concludes the work.

2 Background

2.1 Latency in user interfaces

Latency can be measured as the time it takes for a system to respond to a given input. There are many components contributing to the overall latency of a device. Consider a simplified example of the many steps involved when a user navigates to another view in a smartphone application:

1. User taps on a navigation button
2. Touch is registered by the touchscreen
3. Event traverses operating system drivers
4. Event traverses windowing and UI toolkits
5. Event reaches application which acts upon event, loads another view
6. View eventually loads, graphics subsystem draws new frame
7. Finished frame is flipped to the display
8. Display driver sends new signal to pixels, pixels start updating
9. User sees updated image

Every step in this example introduces latency. For example, the operating system might have other tasks to perform, and thus may not process the touch events instantly. The display has a set refresh rate of usually 60 Hz and can only receive new frames every 1/60th of a second, meaning that flipping the newly drawn frame might have to wait for a short period of time until the next vertical synchronization of the display. The display pixels have a physical response time which means that they will not update instantaneously. All of these example steps add up to form the overall latency of the device.

Fundamental studies on computer systems and response times have been done by Robert B. Miller in 1968. The work cites estimated response times of up to two seconds being acceptable for various tasks where the user expects an acknowledgement from the computer. However certain tasks such as typing requires that the letters show up on the display within 200 ms [42]. In a 1984 study by Shneiderman, sub-second response times are cited as having a positive impact on user productivity [45].

As computing experiences become more and more interactive, our expectations for latency seem to go lower and lower. While the fundamental studies cite seconds as acceptable response times even in interactive scenarios, a more recent study by Ng et al. concludes with an interesting fact they noted in their user tests. After using a custom touch screen with extremely low latency, some subjects reported that the tests “broke” their ability to use everyday touch screen devices, as they now found

the consumer devices' latency completely unacceptable [43]. This indicates that our perception of latency is controlled to some extent by our expectations as well as the latencies we are used to seeing from the devices we use.

A 2011 study by Anderson et al. attempts to find out how much latency in a tablet touch screen is tolerable by the user. Their findings indicate that a significant amount of subjects found latency figures of 580 ms too high for gestures like page turning, web browsing and photo viewing/selecting, however they also note that tablets were new at the time and that this might have skewed the users' perception slightly [24]. Another study by Ng et al. shows that response times of down to 20 ms are needed for the user to perceive the response as instantaneous [37].

2.2 Native applications

Writing applications using each platform's native environment is expected to perform the best and display the lowest response times. Apple and Google have concentrated on optimizing their developer tooling and environments to have applications perform as well as possible on their platform.

2.2.1 Android

The Android ecosystem is vast, and supports various different processor architectures. The latest version of the Android Native Development Kit (NDK r16b at the time of writing) officially supports the following Application Binary Interfaces [1]:

- armeabi (ARMv5, ARMv6 architectures. Deprecated)
- armeabi-v7a (ARMv7-a architecture)
- arm64-v8a (AArch-64 architecture)
- x86 (IA-32 architecture)
- x86_64 (x86-64 architecture)
- mips (MIPS32r1 and later architectures. Deprecated)
- mips64 (MIPS64r6 architecture. Deprecated)

The large amount of supported compilation targets makes it infeasible to develop and ship native code for each third party application. Instead, Android uses Java with its own Dex bytecode format [30]. As of Android 5.0, the Android Runtime (ART) has replaced the Dalvik virtual machine. ART uses ahead-of-time (AOT) compilation at application installation time, in order to compile Dex bytecode into the target platform's native code for improved performance [48]. It is possible for Android applications to ship native code for parts of the application where needed.

2.2.2 iOS

Apple has fewer architectures to target than Google's Android, and so iOS applications consist of binaries with native code instead of bytecode. The main programming languages for iOS applications are Objective-C and Swift [41], both of which are compiled to native code.

According to StatCounter GlobalStats, in April 2018 the iOS operating system had a global market share of 19.23% while the same number for Android was at 75.66%. StatCounter GlobalStats collects this data via web analytics on more than 2 million web sites [10]. IDC has done a similar analysis for 2017Q1, with iOS at 14.7% and Android at 85.0% [7].

2.3 Related popular cross-platform frameworks

It is difficult to find reliable and non-biased data on the popularity of programming languages and frameworks. The TIOBE index attempts to identify the popularity of Turing complete programming languages. They do this by collecting the number of search hits to a query like "+ programming" on popular search engine sites [14]. In the TIOBE index for April 2018, JavaScript placed 8th on the index with a rating of 3.492% and a change of +0.64% from the last TIOBE results [21].

In January 2018 the Stack Overflow developer community website conducted a survey on over 100,000 of its developers. The survey asked about which technologies the participants' are using, as well as which technologies they like most or would like to use [20].

While web frameworks like Node.js, Angular and React were at the top of the most popular frameworks list, it is interesting to note that cross-platform frameworks such as Cordova and Xamarin were popular enough to be included. React places third on the list, but it is unclear whether this includes React Native. Later in this chapter we will see that React and React Native are conceptually very similar, and we can thus assume that a React developer could easily pick up React Native or vice-versa.

Another interesting fact to note from the most popular languages and frameworks is that the majority of them are web technologies. On Stack Overflow, JavaScript has been the most commonly used programming language for six years according to the survey, with over 70% of respondents saying that they use the JavaScript language in their professional work [20]. This is in contrast to the TIOBE index results, where Java, C and C++ have the biggest ratings [21]. Assuming that the TIOBE index has a better view of the global perspective as their methods reach a significantly larger number of programmers, this shows some of Stack Overflow's bias.

In any case though, JavaScript seems to be on the rise both in the TIOBE index results and in the Stack Overflow survey. This data brings an advantage to cross-platform frameworks using JavaScript, as these frameworks will likely be easy to pick up for the increasing amount of existing web developers. The Stack Overflow survey data on most loved, dreaded and wanted frameworks provides another indicator to us on how things might look in the future. Out of the relevant frameworks, React is

ranked at the top of the most loved and wanted technologies. This indicates that it has been received well by many developers, and that it still has a good reputation being a technology that developers want to try out and keep using. In contrast, Cordova and Xamarin are among the most dreaded frameworks that were included in the survey. This means that developers have used the frameworks, but do not wish to continue using them in the future [20].

2.3.1 Apache Cordova / PhoneGap

Cordova is a Hybrid app framework by the Apache Software Foundation. PhoneGap is a distribution of Apache Cordova by Adobe, and for the context of performance benchmarking, these names can be used interchangeably. Hybrid apps enable developers to write mobile applications using the standard web technologies JavaScript, HTML5 and CSS3. The Cordova framework essentially functions by rendering the application as a web site in a WebView component. Through Cordova plugins, the framework exposes JavaScript APIs for access to native features of the platform such as camera or file access [44]. Figure 1 illustrates the high-level architecture of a Cordova application.

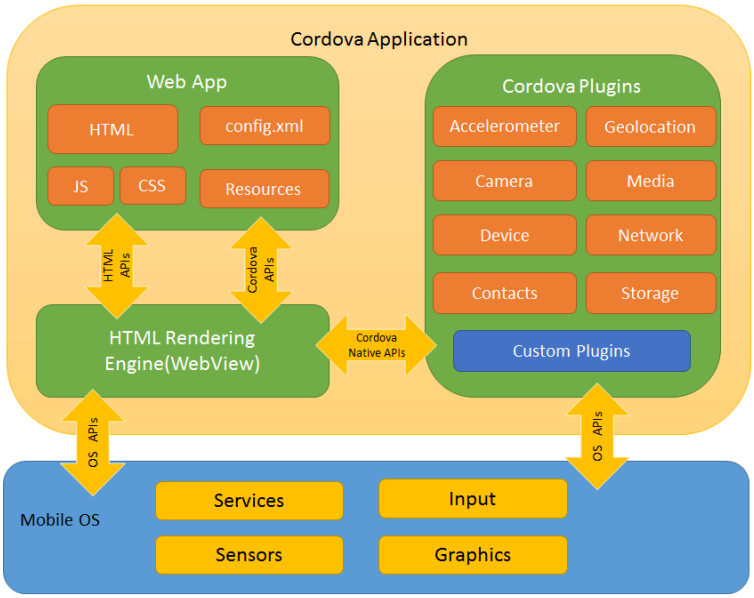


Figure 1: Diagram of a Cordova application's architecture (By The Apache Software Foundation [4])

By being able to use web technologies in mobile development, Cordova developers have the advantage of being able to leverage the enormous HTML/CSS/JavaScript ecosystem of the web platform. Module Counts [11] is a website that collects statistics over various package manager repositories for popular programming languages. According to this website, npm is the most popular JavaScript package manager. As of April 2018, at over 600 000 packages, npm also has the largest number of packages of any platform by over a factor of two. Cordova users can use a large chunk of these packages when writing their applications.

Cordova's downsides include additional overhead of a full-blown web browser, as well as performance issues concerning animations, transitions, responsiveness. As Cordova applications are basically web apps, they will suffer from not having the user experience of a native application. Bosnic et al. [28] and Willocx' et al. [49] benchmarks reveal that Cordova applications use more CPU and memory compared to a native application. Corral et al. [32] have conducted benchmarks testing access times to various device API:s. Their findings indicate that in some cases, access times are up to several orders of magnitude worse in Cordova apps compared to native apps.

Another user experience related downside is that Cordova's UI components have the appearance of web components. There are additional libraries such as the Ionic framework attempting to alleviate this lack of native looking UI components in Cordova. Ionic provides web components, styled with CSS and JavaScript, which aim to replicate the look and feel of native iOS, Android and Windows Phone components [28].

While Ionic improves the situation significantly for Cordova developers, the entire Hybrid app concept is still stuck with the fact that rendering takes place in a web browser. Due to JavaScript's single thread of execution, Ionic's UI code is forced to share computing time with the rest of the application, which will impact UI performance under heavy workloads. Developers have to take special care to implement their animations using CSS3 transformations that support 3D hardware acceleration, which can be limiting and/or difficult for certain types of animations. Regardless if 3D hardware acceleration can be leveraged, animations will still be controlled from JavaScript, which if blocked by other work will cause visible slowdowns and stuttering [3].

2.3.2 Xamarin

Xamarin allows developers to write applications in C#, which can then be compiled into Android, iOS and Windows Phone native code. Xamarin applications are split into two components: one containing the business logic which is shared between platforms and a separate component which renders the UI and has to be written separately for each platform. [29]

Xamarin can be extended with libraries such as Xamarin.Forms which allows for simple UI:s to be built in an XML-based markup language, thus improving code sharing between platforms. Čarapina et al. [31] have developed an application for mobile learning using Xamarin.Forms. They mention that they were able to use readily available Xamarin.Forms components for a large amount of the user interface components, and it was possible to build a specialized drawing component with native implementations for each platform.

Willocx et al. [49] also benchmarked Xamarin in their testing, and compared it to PhoneGap/Cordova and native. Their results reveal that across the test suite used, Xamarin CPU and memory consumption was more in line with the native implementation than the Cordova equivalent. They conclude that Xamarin can be selected over Cordova if the application has a lot of performance intensive code.

2.4 React and React Native

Traditionally in single page web application development it has been the web developer's responsibility to update their application's user interface (UI) to reflect changes in application state. This is done by manipulating the web browser's Document Object Model (DOM) via JavaScript, which results in changes to the web application's UI.

React is a web technology that provides an alternative paradigm to web development by abstracting away the DOM. Instead of manually updating the UI each time application state changes, developers provide React with a mapping from application state to a DOM subtree. In other words, the developer tells React what the application should look like based on the current state. Now to trigger UI updates, developers only need to notify React that the state has changed. React will take care of re-rendering the UI by making the necessary DOM changes.

React manages to handle re-rendering in an efficient manner. Using a feature called Virtual DOM, React can calculate only the needed updates to the DOM for some given state changes [47]. As such, the web browser isn't overloaded with lots of modifications to the DOM each time only some small part of the UI needs to change.

2.4.1 React Components

React encourages developers to split their application into smaller parts called React components. Components are composable, so the developer can build their application from several smaller components which can then be combined together.

React components must declare a `render()` method, which decides what the component will look like when rendered to the DOM. The render method can choose to render HTML elements as well as other React components as child components. React components may contain state, which is isolated from other components. This means that component state can be directly accessed only from within the component itself. The component's `render()` method can make decisions based on the state, and e.g. present a value from the state to the user.

Components can pass along data as well as functions to their child components using properties or "props". This includes passing down component state to child components if needed. In addition, React components may declare special lifecycle methods that React calls at appropriate times. For example, if a component declares a method named `componentDidMount()`, React will call that method after the component has been rendered for the first time (mounted).

In listing 1, we can see two sample React components. The `<Display>` component renders a HTML `<div>` element containing a value supplied by a parent component as props. The `<Counter>` component contains state which maintains the value of a simple counter. This value may be incremented by calling the `increment()` method. The `<Counter>` component renders a HTML `<button>` element which when pressed, calls the `increment()` method. The component also renders `<Display>` as a child component, and passes it the current counter value in the `value` prop.

```
class Counter extends React.Component {
  // Initial state
  state = { value: 0 };

  // Method for incrementing `state.value`.
  // `this.setState()` is provided by React
  // for performing component state updates.
  increment = () => this.setState(state => (
    { value: state.value + 1 }
  ));

  // Render method decides what the UI looks like
  render = () => (
    <div>
      <button onClick={this.increment} />
      <Display value={this.state.value} />
    </div>
  );
}

class Display extends React.Component {
  render = () =>
    <div>{this.props.value}</div>;
}
```

Listing 1: Sample React Components

2.4.2 React Native

React Native builds on the same concepts as React does, but instead of rendering HTML elements, React Native uses the fundamental UI building blocks of the native platform. The end result is that developers can build applications with the platform's native components while writing the application itself in a higher level language, utilizing the concepts and ideas from React.

React Native utilizes threads to run the application and necessary operations. There are two threads doing the main work: The native modules (main) thread and the JavaScript thread [34]. The JavaScript (JS) thread is responsible for running the application's business logic which is written in JavaScript. The JS thread can send/receive messages to/from the main thread in order to draw views and respond to events.

The main thread is responsible for drawing the UI based on commands received from the JS thread. The main thread also handles any native functionality such as making network requests, responding to touch events, accessing peripherals etc. The main thread can send messages back to the JS thread in response to certain events. These messages are sent over a React Native internal component called the bridge.

2.4.3 React Native bridge

React Native works by drawing UI elements on a native thread, with commands sent from JavaScript code. As such there needs to be some means for communication between the threads. This is where the React Native bridge, or BatchedBridge comes in. BatchedBridge is the final barrier between the JS and native threads. Any data that needs to be passed between native code and JavaScript will be sent over the BatchedBridge. This includes drawing any UI elements, information about touch/scrolling events, initiating network requests and receiving results from them etc [16].

We can inspect what is sent over the bridge by enabling a special "spy" mode on the bridge, as in listing 2:

```
import MessageQueue from
  'react-native/Libraries/BatchedBridge/MessageQueue';

MessageQueue.spy(true);
```

Listing 2: Enable bridge spy mode

This will make the BatchedBridge print any messages it carries to the JS console. Here is sample output (truncated to fit on the page, comments added for clarification) from launching a sample React Native application which fetches a list of GitHub repositories, then displays the results in a `<FlatList>` component:

```
// 1. App draws initial layout
JS->N : UIManager.createView([2,"RCTScrollContentView",1,{ ...

// 2. App performs network request
JS->N : Networking.addListener(["didCompleteNetworkResponse"])
JS->N : Networking.sendRequest([{"method":"GET","url":
  "https://.../repositories?q=react%20native&page=1", ...

// 3. App receives data from network request
N->JS : RCTDeviceEventEmitter.emit(["didReceiveNetworkData",
  [68,"<data from network request>"...

// 4. App draws list items based on received data
JS->N : UIManager.createView([7,"RCTImageView",1,{"source":
  [{"uri":"https://avatars3.githubusercontent.com/...
JS->N : UIManager.createView([8,"RCTRawText",1,
  {"text":"author/repo"}])
JS->N : UIManager.createView([9,"RCTText",1,
  {"accessible":true,"allowFontScaling":true, ...
JS->N : UIManager.setChildren([9,[8]])
```

The first column shows which direction the message flowed, `JS->N` means JavaScript to native and `N->JS` is vice-versa.

1. We can see that the JS thread initially requests creating a `RCTScrollContentView` for the currently empty repository list.
2. Eventually the JS thread proceeds to set up network response event listeners, followed by a command for sending the network request.
3. Later, once the network request has completed, the native thread responds by sending the received data over to the JS thread for processing.
4. Once the JS thread has processed the received data, it will proceed by asking the native thread to create image and text components for rendering each GitHub repository as a separate list item.

If we then later scroll the already rendered list we will see events like:

```
N->JS : RCTEventEmitter.receiveTouches(["topTouchStart",
  [{"target":64,"locationX":46.5,"pageY":526.5, ...
N->JS : RCTEventEmitter.receiveTouches(["topTouchMove",
  [{"target":64,"locationX":46.5,"pageY":526, ...
N->JS : RCTEventEmitter.receiveTouches(["topTouchMove",
  [{"target":64,"locationX":46.5,"pageY":523, ...
```

These events are sent from the native thread to the JavaScript thread in case JS e.g. wants to act on scrolling past a certain Y-coordinate in the scroll view. In fact, this test application will use the event to check if we scroll past a certain threshold in the list. If we scroll far enough, the JS thread will eventually trigger another network request for fetching more data (note the `page=2` URL query parameter):

```
JS->N : Networking.sendRequest([{"method":"GET","url":
  "https://.../repositories?q=react%20native&page=2", ...
```

This network request will again eventually complete with some data, which makes the JS thread append the repository list with more items. Note the significant amount of traffic on the bridge even in this fairly simple toy project application. In a larger project, the bridge may become congested, which may result in slowdowns and unresponsiveness. For example a congested bridge may not pass through touch events and UI rendering commands quickly enough, which makes an app seem unresponsive.

2.4.4 React Native component abstractions

React Native components are abstractions of their native counterparts. For the most part they provide a common ground between the functionalities of each platform's native components. React Native components are in some cases platform-specific or have features which are platform-specific. These cases are clearly marked in the

React Native documentation along with which platforms are supported or what kind of behavior can be expected on each platform respectively.

Consider React Native's `<View>` component as an example. `<View>` is a fundamental component for UI layouts, as it supports rendering any number of child components inside it, even other `<View>` components. The `<View>` component can be styled, meaning we can alter its size, shape, color as well as placement of child components within. As per React Native documentation, a `<View>` is an abstraction of the native platform's equivalent of a view component. On Android this is an `android.view`, on iOS it's `UIView` and the HTML equivalent is `<div>` [33].

If we dig deeper into the React Native View documentation, we can see that `<View>` components are customized by setting various props on them. We can see that many props are supported on both Android and iOS, for example `accessibilityLabel` can be set to override what a screen reader will call the component on both Android and iOS. However, e.g. `accessibilityTraits` is an iOS-only feature, as this functionality only exists on the iOS platform [26]. Special care must be taken when using functionalities that are not supported across all platforms. Namely, the developer needs to ensure that the application behaves well also on platforms where the functionality is not supported.

2.5 Related work

2.5.1 Evaluations of hybrid app frameworks

Most similar studies that we were able to find at the time of writing have been performed on more traditional hybrid app frameworks such as PhoneGap and Cordova and web applications. This is good for validating the methods used.

Corral et al. [32] have conducted a performance comparison between PhoneGap and the Android native platform. By writing test applications that test various hardware, data and network accesses, they were able to time identical operations between the native and web apps. They conclude that the PhoneGap application is less performant compared to the native application. However they also note that the total impact depends a lot on the nature of the application.

Willocx et al. [49] tested response times, CPU usage, memory consumption and disk space requirements of native, PhoneGap and Xamarin. They conclude from their results that cross-platform tools in general do introduce performance penalties. However they also mention that the impact is often times acceptable, especially on high end devices.

Willocx' et al. tests were conducted by instrumenting an open source application called PropertyCross, which is implemented in multiple cross-platform frameworks to aid developers in choosing which framework to use. Unfortunately the PropertyCross project is no longer maintained [15], and does not contain React Native implementations which would be useful for this study.

Some research exists on the network performance of cross-platform frameworks. Yun Ma et al. [39] have focused on testing how native applications perform HTTP requests compared to equivalent web applications. Their findings indicate that web

apps may outperform native apps in a surprising number of cases. Occasionally the web browser is able to leverage newer protocols and underlying connections from already requested resources where the native apps would need explicit code that accomplishes the same feats [39].

2.5.2 Evaluations of React Native

Not much research seems to exist on React Native specifically yet.

Axelsson et al. [27] have performed an evaluation on the development process of React Native, and additionally tested the user experience of a React Native application as compared to a native application. They found that while users would notice a difference when comparing the applications side-by-side, users were not able to spot a difference when testing the applications in isolation. Axelsson et al. also found that in many cases, the development time of a React Native application can match, or even beat, that of one native platform's corresponding application.

The React Native framework's performance has been analyzed from a resource usage perspective by Andersson et al [25]. They conclude that React Native often requires more resources compared to a native Android application. However they also conclude that this may be an acceptable tradeoff when considering the ease of development of an application for both Android and iOS using React Native.

Another study examining the development cycle of a React Native application has been done by Majchrzak et al. [40]. They compare the React Native development cycle to a few hybrid app frameworks. The work notes that extensive study has yet to be performed on these frameworks. The findings of the work indicate that React Native is at least as feature-packed as its contenders, and that React Native has the biggest community at the moment out of the evaluated frameworks.

2.5.3 Similar performance evaluations

Y. Gao et. al. [35] have implemented a system called DRAW which aims to reveal UI performance problems in an application such as excessive overdraw and slow UI components. DRAW is made possible by instrumenting the View component of the Android operating system. This makes it possible to see really detailed data on what is happening deep in the Android internals, even better than what Android's debugging tools can achieve.

2.6 Scope

This thesis will focus on comparing React Native vs native Android implementations, as well as compare React Native components with each other. We chose Android due to more personal experience and a larger market share compared to iOS. We will thus leave out comparisons between React Native and iOS native implementations.

Ideally we would instrument the Android View component similar to what the DRAW system by Y. Gao et. al. does [35]. However doing so requires significant amounts of development resources as well as flashing custom firmware onto phones, neither which were possible within the scope of this study. We will instead focus

on finding other, higher level ways of taking the measurements and performing the benchmarks.

3 Methods

Data acquisition happens with the aid of test applications specially built for this purpose. The test applications will benchmark various scenarios and collect analytics data to the extent possible for later analysis. We can instrument our test applications in various ways and make them collect data such as timestamps of various events occurring. We then take these timestamps, compare them and compute the time it took for all operations between the timestamps to complete.

We also collect measurements as well as validate our methods by examining system logs, systrace captures, high-speed camera recordings, Android Studio's profiling tool, and using dumpsys gfxinfo to retrieve framerate information.

All tests will be performed on release/production builds of the test applications to eliminate any extra debugging features and their potential performance impact. Testing will be performed on a LG Nexus 5 phone, running stock firmware (Android version 6.0.1). The phone is plugged in via USB to a PC laptop during the tests at all times.

This particular phone was selected because it runs an unmodified version of Android, and thus does not contain any additional software that may interfere with the phone's performance. The LG Nexus 5 is also an older high-end device. Released in October 2013 it is outperformed by today's high-end devices. This will amplify any potential performance issues with React Native, and make visual inspection easier. Occasionally when relevant, we will test with a Huawei P20 Pro phone in addition to the LG Nexus 5 phone. The P20 Pro is a 2018 flagship device which will give us an insight into what the results will look like on a more modern device with better performance.

3.1 Measuring application launch

Android ships with tools for measuring application launch times. The Android developer guides define application launch time as the time it takes from launching the application process until the application has been drawn for the first time [9]. They note that the Android system, or more specifically the ActivityManager component will by default log all application launch times to the Android system log, logcat. This information is accompanied by a millisecond accurate timestamp, which also applies for all other log entries in logcat. The ActivityManager timestamp measures the time from launching an activity until the activity has finished its initial draw pass [9].

The developer guides mention that the launch time measurements do not necessarily measure all application elements, and as such the time it takes for some resources to load may be left out. Only the time it takes to render what has been defined in the application's layout file, and the time it takes to render elements created during application initialization is taken into account [9]. This is good to keep in mind when we will be comparing a native application with a React Native application, as React Native applications have no means of providing an initial UI through JavaScript yet at this stage.

This thesis measures the time it takes for a simple app to become usable after being launched. In the case of a native application, the ActivityManager launch timestamp log entry will be sufficient, as this means the application has already had the chance to draw its basic components. However in the React Native case, the screen is still blank at this point.

We can observe the following logcat output while launching a simple Hello World React Native example application:

```
05-03 09:13:18.472   791   801 I ActivityManager: START u0
    {flg=0x10000000 cmp=com.launch/.MainActivity} ...
05-03 09:13:18.488   791  1305 I ActivityManager: Start proc 6659:
    com.launch/u0a86 for activity com.launch/.MainActivity
05-03 09:13:18.709   791   809 I ActivityManager: Displayed
    com.launch/.MainActivity: +231ms
05-03 09:13:19.112  6659  6685 I ReactNativeJS: Running application
    "launch" with appParams: {"rootTag":1}. __DEV__ === false,
    development-level warning are OFF, performance optimizations are ON
```

From the output we can see that there is a brief moment after the ActivityManager launch timestamp logs until React Native has started running the JavaScript application which will render the application's UI. Android does not have any idea about the application's components or layout until the JS code is running and has sent commands to create these components over the bridge. Suffice to say, the application won't be usable until after this point in time.

Instead of only using the ActivityManager's app launch timestamp for React Native apps, we will factor in how long it takes for the React application to call its root component's `componentDidMount()` method. This will give us a comparison between when a native application would become usable (ActivityManager timestamp) versus a React Native application becoming usable (`componentDidMount()`). We will also measure how long it takes from app start until our application's JavaScript bundle starts running.

3.2 Benchmarking React Native components

We have means of instrumenting our applications purely from the JavaScript side to perform simple timing tests and benchmarks. These can be useful for measuring launch times, as well as when components have finished rendering. There are some disadvantages to logging from JavaScript though, as there are potential performance impacts when logging large amounts of data. There are also issues with the accuracy of timers, as well as uncertainty to when code runs due to JavaScript's event-based model.

3.2.1 Problems with console logging

Logging has one significant disadvantage per the React Native documentation, namely it decreases performance [34]. We have to work around this by restricting the amount

of console logging we do. Also we have to take extra care and log to console only after results have been gathered, not during test where we would otherwise skew the results.

Another feature greatly impacting performance is React Native's development mode. In development mode, React Native performs a lot of extra work to make the debugging experience better [34], thus reducing performance compared to production mode. This means we need to run our test applications in production/release mode whenever performing the benchmarks, or else the results won't accurately reflect a real application running in production mode.

Yet another complication that affects our results is that viewing the console logs is most convenient when enabling a feature called remote JS debugging. However, this causes the JavaScript to run in a browser or our computer, which skews the results as the computer will likely be way more powerful compared to a mobile phone. For our testing, this means that we have to find alternative ways of gathering the logs.

Luckily for us there are alternatives, as React Native also writes console logs to the phone's system logs, we have ways of reading the logged data without incurring significant performance penalties. On Android we can use "adb logcat" for viewing system logs, and on iOS there is a "View Device Logs" feature in Xcode that will show us the phone's system log.

3.2.2 Collecting timestamps from JavaScript

Purely doing our benchmarking in JavaScript means we will suffer from timer inaccuracy. The built in JavaScript `Date` object provides a `Date.now()` function which unfortunately only gives us millisecond accuracy at best [13]. Modern browsers provide `window.performance.now()` which when called provides timestamps of up to microsecond resolution [13]. Node.js has `process.hrtime()` for getting a high-resolution timestamp with nanosecond accuracy.

Unfortunately for us the React Native runtime has neither of these more accurate timing functions. This is unfortunate because the millisecond resolution of `Date.now()` is not good enough for our purposes. At 60 frames per second, rendering one frame takes only 16.7 milliseconds. One millisecond is already a significant portion of that time.

Upon further investigation there seems to be an undocumented `global.nativePerformanceNow()` function in React Native. Digging into the source code of React Native reveals some further details about this undocumented function. On Android, `global.nativePerformanceNow()` will call the native (NDK) function `clock_gettime()`, which has nanosecond resolution [5]. And on iOS the native function called is `CACurrentMediaTime()` from the iOS Core Animation API, which per the documentation bases its result off of `mach_absolute_time()` which in turn has nanosecond resolution. We will use the `global.nativePerformanceNow()` function whenever taking timestamp measurements in JavaScript code.

3.2.3 Comparing React Native components

React Native contains several implementations of some components such as lists, where each different component might have a certain use case or platform where it works best. The following methods can be used for benchmarking React Native components:

- Measuring render latency. We can do this by timing how long it takes from informing React Native it should render the component to the component (or its children) having their `componentDidMount()` lifecycle method called. We can use the more accurate timer method as mentioned above for increased timer accuracy.
- Measuring the amount of traffic across the bridge, with the rationale being that less traffic results in a more responsive application.
- Visual inspection. For example, if a list is not able to render items fast enough, the list items will be blank for a brief amount of time.
- Measuring the framerate of the JS thread, telling us under how much pressure the JavaScript engine is. This can also tell us how much processing of events gets delayed due to the JS thread being overloaded with work.
- Using performance debugging tools of the native platform to capture information such as UI framerate, CPU usage, memory usage.

3.3 Native Android vs React Native components

React Native is often used to build standalone applications that are built up from scratch using JavaScript and React Native components. However, it is also possible to embed React Native into an existing native application, and write only a subset of your application in React Native [23]. In fact when you start a new React Native project, you end up with a native application whose only functionality is wrapping a React Native application.

We can leverage this integration possibility for our benchmarks. By writing our benchmarking routines in native code and embedding React Native into our native application, we can use all tools and API:s that the native platform makes available to us. Recall that React Native works by drawing native UI components according to some commands sent to it by a JavaScript engine. Using native code, we should be able to take note of when React Native performs its UI layout/draw commands, and compare that to when a corresponding native UI implementation performs the same commands.

We now need to find some UI rendering related event which is generic enough that it can be hooked to both for our embedded React Native UI, but also for an equivalent native UI implementation. We can then measure the time it takes to render a view after commanding the application to start drawing the view. This test will be valid on both platforms, and should give us very interesting results which we

can use to compare directly the performance of a React Native UI implementation vs. a native equivalent.

Doing all of the measurements programmatically also brings the added advantage of automating our benchmarks. This in turn makes it trivial to take a large number of samples and thus get more accurate results by averaging these samples together. Furthermore, recording lots of samples minimizes effects of possible background tasks, CPU frequency scaling due to increased CPU load and/or recent interaction with the device, and other temporary interfering factors.

It is useful to do logging to console in both our native and React Native implementations, as Android provides millisecond accurate timestamps in the logcat output. We can use this to easily capture events occurring at different stages of drawing the UI. Without deep expertise on the respective platforms' internals, we can still verify these timestamps for validity through means of visual inspection.

3.4 Visual inspection

Some results will be verified for validity by recording manual interactions with the device captured with a 240 FPS slow motion video camera. As this is four times the target framerate of the phone's display, this method gives us enough accuracy to record four data points per output frame of the phone.

When measuring application launch, we validate our automated results by recording a user manually starting the test application. Here we will look for the time from when the user finishes their tap gesture on the application icon (finger is lifted from the screen) until the time the initial user interface is fully visible on the screen. Our test applications draw a black screen as their initial UI, so we end the measurement when the entire display is black.

When measuring navigation times, we validate the results by recording a user manually tapping a button that triggers the navigation action. The screen we navigate from has a white background, and the screen we navigate to has a black background. As soon as the entire display is black we conclude the test.

List scrolling will also be tested manually, and the behavior of the scrolling list is an important indicator to monitor closely. If no problems can be detected during manual scrolling of a list, even with the use of a high speed camera, then we can conclude that the particular list implementation is fast enough.

We validate the methods used for comparing render latency between Android vs React Native components with a high speed camera. Our setup consists of a test application with a button, which when clicked will render a list of components and measure the total rendering time of these. The verification consist of capturing a user manually tapping this button, and measuring how long it takes from this point until the list of components appear on the device's screen.

4 Implementation

4.1 Application launch

Implementing a test application in this case is trivial, as the only thing the test applications will do is to draw a view with a custom background color. A small modification is needed to the React Native application, where we implement a `componentDidMount()` method that simply logs any easily identifiable string to the console. We do the same for measuring when our JS bundle starts running by placing a `console.log` statement into the global scope (see listing 3).

```
console.log('JS is running');

export default class App extends React.Component {
  componentDidMount() {
    console.log('launch measurement completed.');
  }
  render() {
    ...
```

Listing 3: Sample component which will log when it has finished mounting

Since we can use the timestamps directly from logcat, we do not need to worry about capturing timestamps from JavaScript code. The results are in the hundreds of milliseconds, so the millisecond accuracy we get here is sufficient.

The measuring part is slightly more involved, and could require a lot of manual work to get accurate measurements. Instead we write a shell script that will use the Android Debugging Bridge (adb) for commanding the test phone over USB. We can automatically launch our test application, wait until it has launched, then kill the application so that we can be sure it is no longer running in the background. We can repeat this as many times as we want, and when we are done simply read the timestamps from logcat to obtain results. The shell script used can be found in Appendix A. The shell script allows substituting the `APPLICATION` variable to launch and measure any installed Android application. The results are presented in Section 5.1.

4.2 React Native components

4.2.1 Measuring bridge traffic

We briefly touched upon monitoring the bridge traffic earlier in the background section. There we viewed the bridge traffic by means of logging it to the console. It is also possible to attach a callback function to the bridge, enabling us to run custom code for each bridge event. This in turn gives us a handy way of measuring the amount of traffic passing over the bridge.

Listing 4 demonstrates how we can inspect the bridge traffic with a custom callback and count the number of messages passing through it each second. The upper half of the code snippet will gather all bridge events into the `events` array as they arrive.

```
import MessageQueue from
  'react-native/Libraries/BatchedBridge/MessageQueue';

// Gather events
let events = [];
MessageQueue.spy(info => {
  events.push(info);
});

// Log time elapsed & number of events. Runs every second.
const startTime = new Date().getTime();
setInterval(() => {
  const elapsedTime = (new Date().getTime() - startTime) / 1000;
  console.log(elapsedTime, events.length);
  events = [];
}, 1000);
```

Listing 4: Sample code for logging amount of bridge traffic

The lower half of the code snippet in listing 4 runs once every second using a timer. Here we count the number of accumulated events in the `events` array, followed by clearing the array. We log the amount of events we counted, as well as time elapsed since the test started. This way we end up with data containing a timestamp and the number of bridge events that occurred during that time.

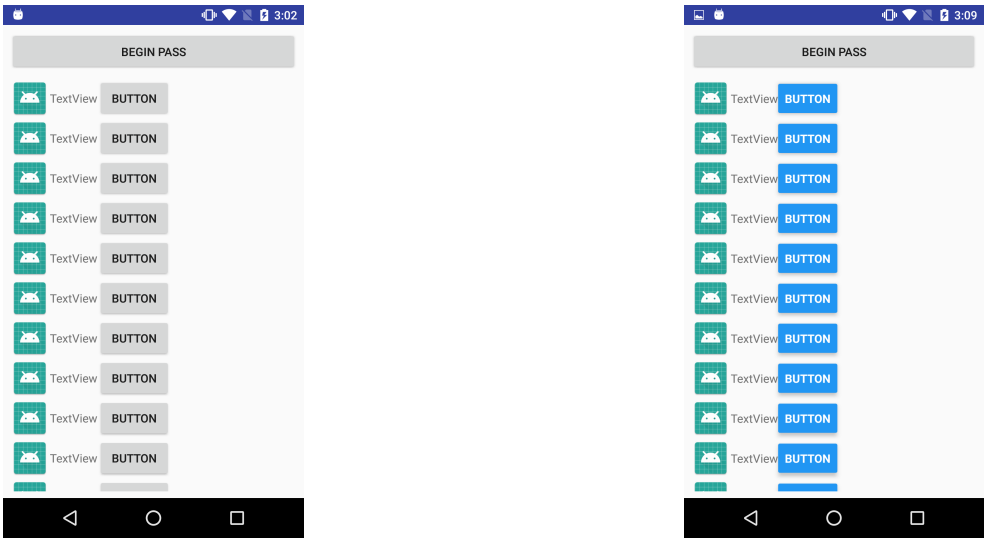
4.3 Native Android vs React Native

4.3.1 Fundamental components

For this test, the following fundamental UI components were selected for comparison between Native Android and React Native applications:

- TextView component
- Button component
- ImageView component showing a local image

These were selected due to being common components in mobile apps. We draw these three components on the same row, and can vary the amount of rows for the different tests.



(a) Native Android components

(b) React Native components

Figure 2: Test application for benchmarking component render times

The test application can be seen in Figure 2 in both Native Android and React Native configurations. The application always renders a Begin Pass button, which when tapped will render a configurable amount of component rows below it. The time it takes to render the components will be measured using a method described hereafter, and following the measurement all test components will be hidden from view again to prepare for the next test pass. In order to simulate user presses we use adb in a a shell script to automatically tap the Begin Pass button at regular intervals.

An interesting interface was found on `ViewTreeObserver.OnGlobalLayoutListener`. As per the Android documentation [22], this specifies a callback which will be called whenever the layout of the view tree changes. This means we will be able to tell when the layout pass of the Android rendering pipeline has completed. This is very valuable, as it applies both for native and React Native code, and happens rather late in the pipeline as we can see from Figure 3.

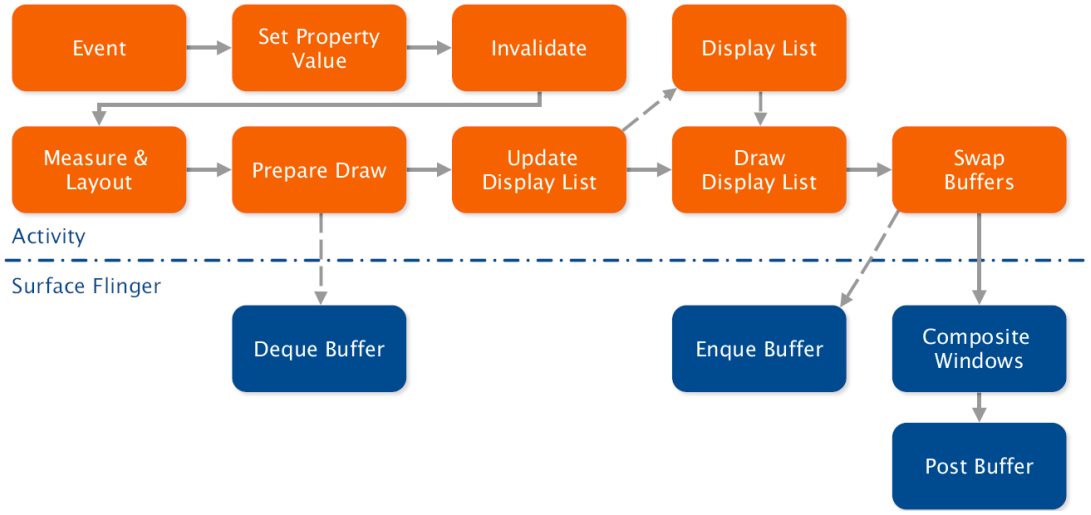


Figure 3: Android Graphics Pipeline (M. Garbe [2])

The only step after layout computations will be drawing [6], and this should be very similar either way both for our native / React Native examples (which we verify using systrace, see Figures 4 and 5). Furthermore, through a combination of logging and visual inspection we were able to determine that the timing of the GlobalLayoutListener callback closely matches with when new content is visible on screen. This is true even when rendering complicated views that take seconds to calculate a new layout for. This indicates that the measure and layout pass takes up the majority of work in the graphics pipeline, with the rest of the rendering steps happening very fast. This discovery helps make the GlobalLayoutListener a valid hook point to measure against.

We will attach a callback as follows on the root LinearLayout Android component of our app:

```
layout.getViewTreeObserver().addOnGlobalLayoutListener(() -> {
  if (testing) {
    endPass();
  }
});
```

Listing 5: Adding OnGlobalLayoutListener to a LinearLayout

The `testing` helper variable in Listing 5 stores our benchmark state and whether we are currently performing a test pass or not. This is needed because the GlobalLayoutListener will get called outside our test as well, such as when the view is initially rendered during app startup, or when we remove the components from the view after our measurements.

When the Begin Pass button is pressed, we store the current system time in nanoseconds and set the `testing` variable to true. Then depending on the test configuration, we either add the native components to the layout ourselves, or send

an event to React Native signaling that it should draw its components into the view. At this point we simply need to wait until the OnGlobalLayoutListener callback fires, at which point `endPass()` will get called where we can compute the elapsed time.

On the JavaScript side, we have several additional hooks that are interesting for our React Native tests. We instrument the following parts of the React Native application (in addition to the GlobalLayoutListener hook on the native side):

- `render()`
- `componentDidMount()`

These will provide insight into any possible additional delays in the JavaScript execution, and a more detailed overview of exactly which parts use up extra time. These methods are instrumented by making them log an easily identifiable string to console. We can then measure how long it takes for the component to reach the different stages from the start of our measurement, and finally also include the GlobalLayoutListener timestamp in the results for comparison to the Native Android configuration.

This test simulates very closely the situation when the user presses a button on the screen, and some views get rendered or modified as a result of that. The observant reader might be concerned about extra overhead from the event passing to JavaScript when comparing to the native test, however this is very much how a normal React Native button works. Even in normal usage, the button is rendered as a native component by React Native. React Native has set up an onClick listener on the button, which when triggered, will forward the native click event to the JavaScript code over the bridge. This is essentially what we are doing in this test.

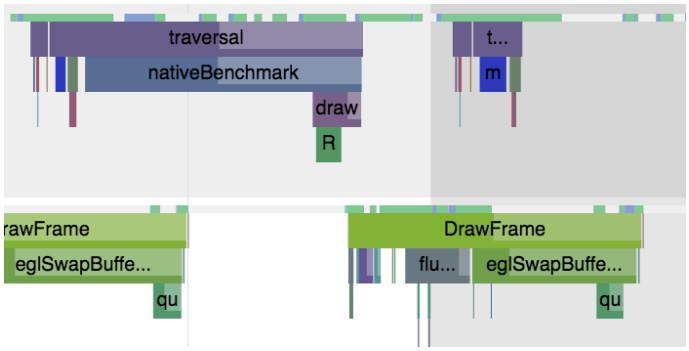


Figure 4: Benchmark of a native component viewed in systrace

Figure 4 shows a sample systrace capture from running our benchmark application on a native Android UI component. Here we can see which parts of a frame the benchmark uses for timing, labeled with `nativeBenchmark`. The benchmark pass starts as soon as the previous frame's measure/layout step has completed, and ends with this frame's measure/layout step which also marks the beginning of the GPU starting to do drawing work.

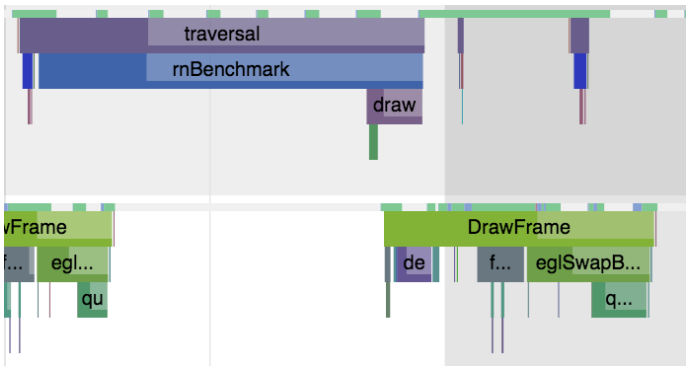


Figure 5: Benchmark of a React Native component viewed in systrace

Figure 5 shows a sample systrace capture for drawing a React Native component. Here the part used for benchmark timing is labeled with `rnBenchmark`. We can see that it is very similar to the native component systrace capture.

4.3.2 Navigation

React Native only contains navigation facilities for iOS using the NavigatorIOS component which is built on top of iOS' UINavigationController [12]. This won't work on Android, and is thus problematic for cross-platform applications. Alternatives exist in the form of third party libraries, and a popular React Native navigation library at the moment is React Navigation. Facebook recommends this library on the official React Native documentation, and it provides easy cross-platform navigation facilities. It is built entirely in JavaScript, and is very customizable [17].

React Navigation provides a Stack navigator, Tab navigator and Drawer navigator cross-platform components for React Native applications. These imitate the look and feel of the corresponding native navigators, and can be nested in each other to accomplish the wanted navigation functionality. For example a Tab navigator might act as the main navigator of an application, allowing the user to browse through different screens by swiping or tapping tabs on an always visible tab bar. However by using the tab navigator together with a stack navigator, a button on one of the tab pages might launch another view which overlays the whole screen. On Android, stack navigation is realized by launching different activities of an application. Swipeable tabs can be implemented with a ViewPager widget, and there's a DrawerLayout for implementing a drawer navigator.

We will be testing a basic stack navigator in both scenarios. Both our test applications will feature two screens, with the first one containing a button for navigating to the second one. The first screen background is white, and the second screen background is black to aid visual inspection.

In the Android application, we log to console when the user lifts their finger from the button. Immediately after this happens, we signal the system to start the second activity. We disabled the transition animation which by default creates a smooth animation between the activities. This will help visual inspection in determining the exact frame the new activity has finished drawing. Furthermore, the transition was visually determined not to be significantly different between the platforms, so we

did not study the transitions further. We log to console when the second activity's `onCreate()` method was called.

The React Native application is similar, we also log when the user lifts the finger from the button. This triggers a React Navigation `navigate` action, which will cause the next screen to be rendered. The React Navigation stack navigator also has a transition animation by default, which tries to imitate the corresponding animation on Android. We can disable it to aid visual inspection by passing a custom `transitionConfig` when creating the stack navigator. We log to console when the second screen's `componentDidMount()` method was called.

5 Results

5.1 Application launch

In this test we compared app launch times to find out how much overhead there is in launch times for React Native. Figure 6 shows the timing results of starting a simple native Android application (above), as well as a React Native application (below). The leftmost graph shows the results from a LG Nexus 5 phone, which was a flagship device announced in 2013. To the right we see the same tests performed on a 2018 flagship phone, the Huawei P20 Pro.

The ActivityManager timestamp is relevant in both native and React Native cases. The ActivityManager timestamp measures the time from activity launch until initial render, however with React Native apps there was still some extra work to be done before the application is in a usable state. Regardless, the results for the ActivityManager timestamp are very similar in all test cases, interestingly enough even between the different phones given their very different performance characteristics.

After the ActivityManager timestamp, the Native Android application has finished rendering and is now usable from this point. However the React Native application remains blank, as it has not yet ran the JavaScript application and thus does not yet know what to render. Without making changes to native code, the default initial view is a blank white screen.

The React Native application remains in an unusable state until all of the steps visible in Figure 6 have been completed. The first additional step compared to a native Android application is starting the JavaScript runtime, which is represented in the figure as "JS timestamp". This is the time it takes to start the runtime and parse our JS bundle, until our application's JavaScript code starts running.

After the JS runtime is up, the JavaScript code needs to render the initial React Native view. The `render()` function is called at the end of the `render()` timestamp, which declares the initial view. Finally, `componentDidMount()` gets called as a sign of our initial view having successfully rendered.

These results were confirmed manually by using a 240 FPS slow-motion camera. We timed how long it takes from manually launching the app (finger leaving the touch screen) until the application has rendered the black background. We found the results from our manual testing to match with the automated testing.

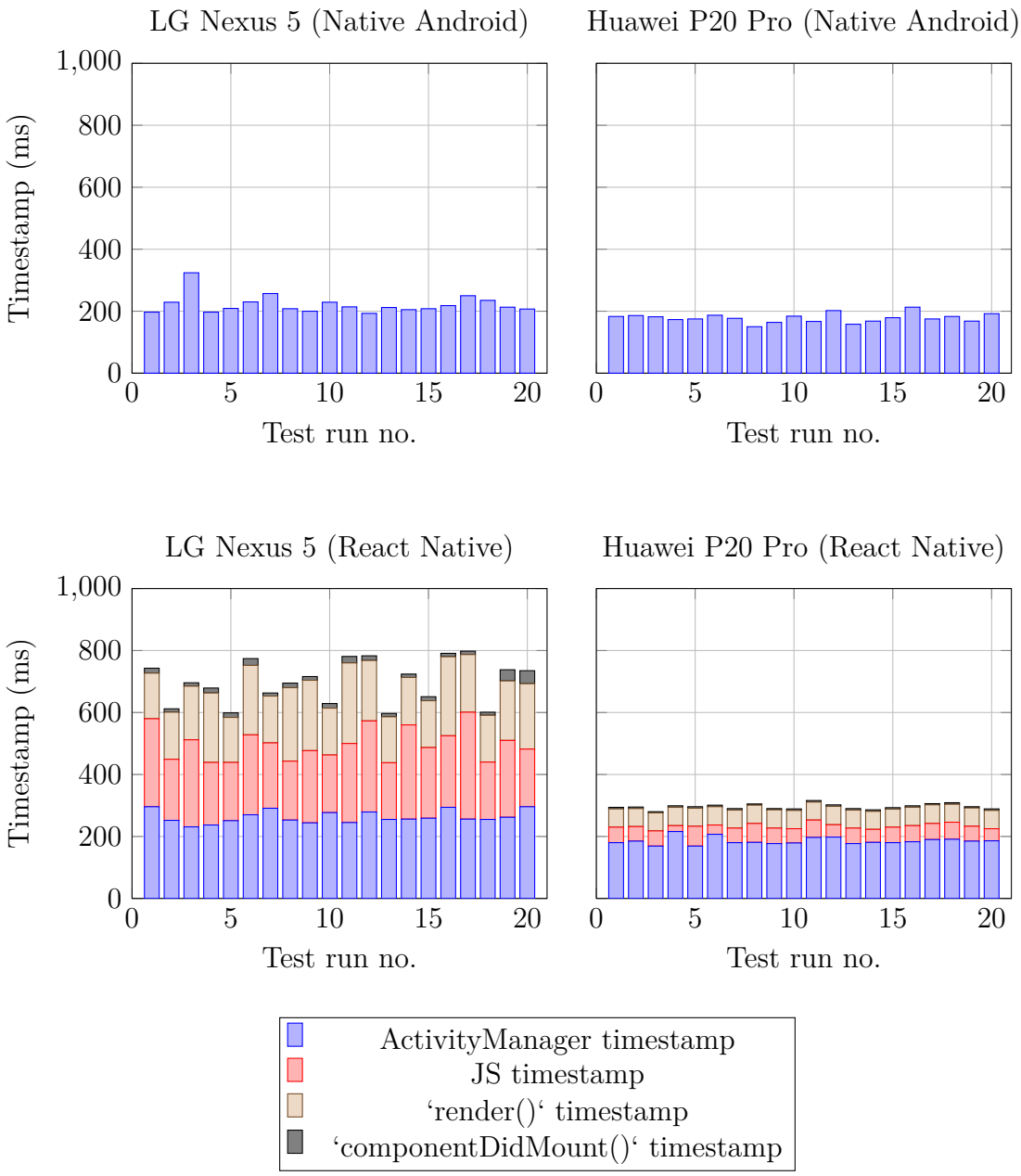


Figure 6: React Native app launch times

	Android (LG)	RN (LG)	Android (Huawei)	RN (Huawei)
$\overline{x}$	221.8	700.3	178.3	296.3
σ^2	29.6	68.8	14.5	8.6
min	193	597	150	280
max	324	798	213	316

Table 1: Total time to layout in application launch

5.2 React Native components

5.2.1 List components

For this test, we compare React Native components `<ScrollView>` and `<FlatList>` against each other. Our test application inserts 1000 items into the list, each item containing a 256x256 pixel image and a text next to the image. The items are 64 pt tall. The test application automatically scrolls the list at a rate of 1000 pixels four times each second, totaling a rate of 4000 pixels/s, or about 63 items each second. In Figures 7 we plot how many events are passing through the React Native bridge over time in each case.

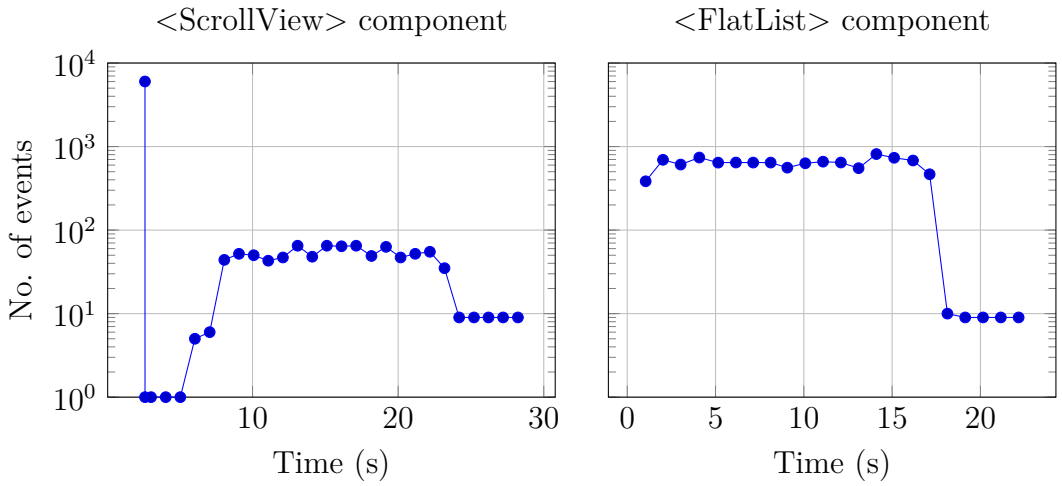


Figure 7: React Native list components, scrolled at 4000 pixels/s. (Note the logarithmic y-axis)

	<ScrollView>	<FlatList>
$\overline{x}$	247.1	490.1
σ^2	1130.5	281.2
min	1.0	9
max	6014.0	813

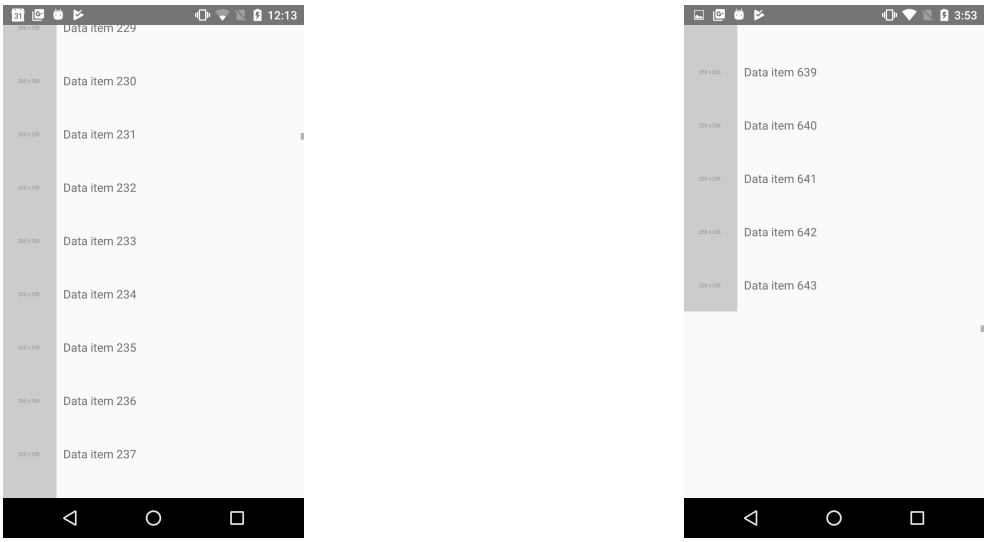
Table 2: Number of events while scrolling list components at 4000 pixels/s

In the `<ScrollView>` test we can see a huge spike of up to 6000 events when first running the app. This is because a ScrollView will render every single child element regardless of whether they are visible on the screen or not. This caused React Native to send commands for drawing all 1000 items at once, resulting in the JS thread being completely locked up for several seconds while all the commands were being sent to and processed by the native thread.

This meant the app remained unresponsive until the first messages were being passed through the bridge again, which occurred around 7 seconds after app launch. After that however, the list scrolled very smoothly and all items were rendered

without any problems, with around 60 messages being passed through the bridge every second. At around 23 seconds the list had scrolled all the way to the bottom and the test ends.

The `<FlatList>` test showed how FlatList manages to spread out the bridge messages more evenly compared to ScrollView. There is no huge initial spike like in the previous test, and while scrolling the number of events passing through the bridge remains fairly constant. However, FlatList suffered from some problems at sufficiently high scrolling speeds such as in this test. Occasionally FlatList on our Nexus 5 test device could not keep up and the items were not being rendered in time, resulting in empty slots briefly appearing in the list as can be seen in figure 8b.



(a) FlatList rendering all items

(b) FlatList with missing items

Figure 8: At high scroll speeds, FlatList was not able to render items fast enough

Third party list components: Third parties have also implemented custom list view components which aim to bring better performance or features in certain use cases. RecyclerListView by Flipkart is one such component [18]. RecyclerListView works by reusing individual list elements through a process called "recycling". This is similar to how Android's native RecyclerView works.

Instead of freeing and reallocating resources each time a list element is scrolled off-screen, RecyclerListView will reuse list elements by moving them to the opposite side of the screen, then only replacing the data contents of the list elements. Re-rendering is faster due to not triggering expensive layout updates, and the reused list element can be scrolled into the view faster. This type of optimization works well for huge lists where the data is mostly the same, and the list elements do not vary much in dimensions. Facebook has likely focused on making the ListView and FlatList components more generic, thus no recycling is used in these [19].

An impressive feat of RecyclerListView is that it is truly cross-platform and implemented only using JS. Flipkart claims that it even works in web browsers.

RecyclerListView is implemented using React Native's ScrollView component, and custom layout code to place the list elements into the ScrollView according to the current scroll position [18].

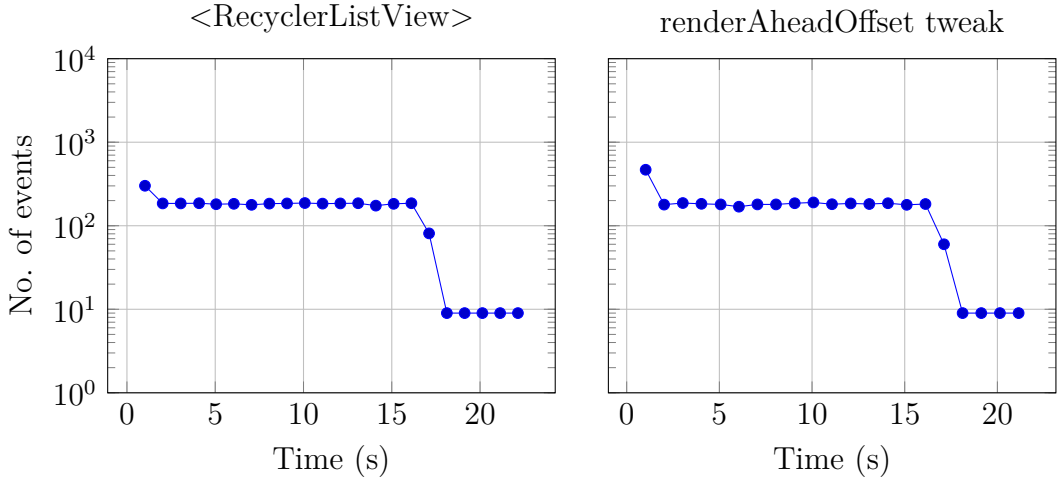


Figure 9: Flipkart's <RecyclerListView> component, scrolled at 4000 pixels/s. (Note the logarithmic y-axis)

	<RecyclerListView>	renderAheadOffset tweak
$\overline{x}$	144.5	156.8
σ^2	82.6	100.8
min	9	9
max	301	468

Table 3: Number of events while scrolling <RecyclerListView> at 4000 pixels/s

RecyclerListView manages to drastically improve performance in this particular benchmark compared to both the naive ScrollView implementation and FlatList. Figure 9 shows that the bridge traffic is significantly lower than when using either of the built in list components, but through visual inspection we could identify some remaining problems with empty list items during scrolling and items not being rendered quickly enough. This was easily solved by increasing the `renderAheadOffset` prop value. The second graph in Figure 9 shows that this tweak did not cause any other changes in bridge traffic than a slightly larger initial spike (468 events compared to 301).

With the `renderAheadOffset` tweak in place, no performance problems were visible even through manually scrolling the list as fast as we possibly could. This meant we could not tell the list apart from a native Android RecyclerView anymore in this test.

5.3 Native Android vs React Native

In these tests we compare the time from the creation of a component until the component's layout pass has completed. We test with image, button and text components, which we group into a configurable amount of rows.

5.3.1 Fundamental components

1 and 10 rows of components: We begin by drawing 1 row of each component to establish a baseline. This will show how much time each platform spends on drawing a very simple view with just the three components from that one row. We then proceed to testing with 10 rows of components. This is done to amplify any potential performance issues and make these easier to detect. 10 rows of components makes for a total of 30 components, which is a reasonable maximum number of components we would expect to see on the screen at once.

The results in Figure 10 shows that with reasonable amounts of components, both apps remain responsive. The results on Android are very consistent, while on React Native there are minor fluctuations. It is worth mentioning that in the trivial case with one row, the results on Android are very consistent around the time it takes to draw one frame at 60 FPS, or 16.67 ms. The results never go below this value either, which leads us to believe that the minimum value is limited by vertical synchronization. We speculate that Android polls for input events at the same rate as VSync, which would trigger our view drawing code when a frame has been drawn. Since this is such a trivial case for a native application, rendering the components themselves is very quick. Our GlobalLayoutListener callback then gets called while rendering the next frame, resulting in a measurement just slightly above 16.67 ms.

The 10 row case on Android has more variation. We suspect this is because the results are no longer bounded by the 16.67 ms lower limit, as rendering a screen full of components no longer can be done within the one frame deadline on the older LG Nexus 5 phone.

The React Native results are interesting. We see consistent fluctuation here compared to the Android results, even within the different tasks that we were able to measure from our JS code. It is important to remember that the bars show the time from the previous task until the end of the task represented by the bar. There may be extra work involved that does not belong to the presented task itself. For example, the GlobalLayoutListener callback never gets called faster than in the Android examples, but it occasionally takes much longer suggesting there is some extra work going on there apart from Android working on the components' layout.

We speculate that these inconsistencies stem from the fact that React Native has to deal with a JS runtime complete with garbage collection, an event based nature and other sources of timing inconsistencies. It is also possible that the JavaScript code is more sensitive to background tasks compared to native Android UI code, resulting in larger latency fluctuations.

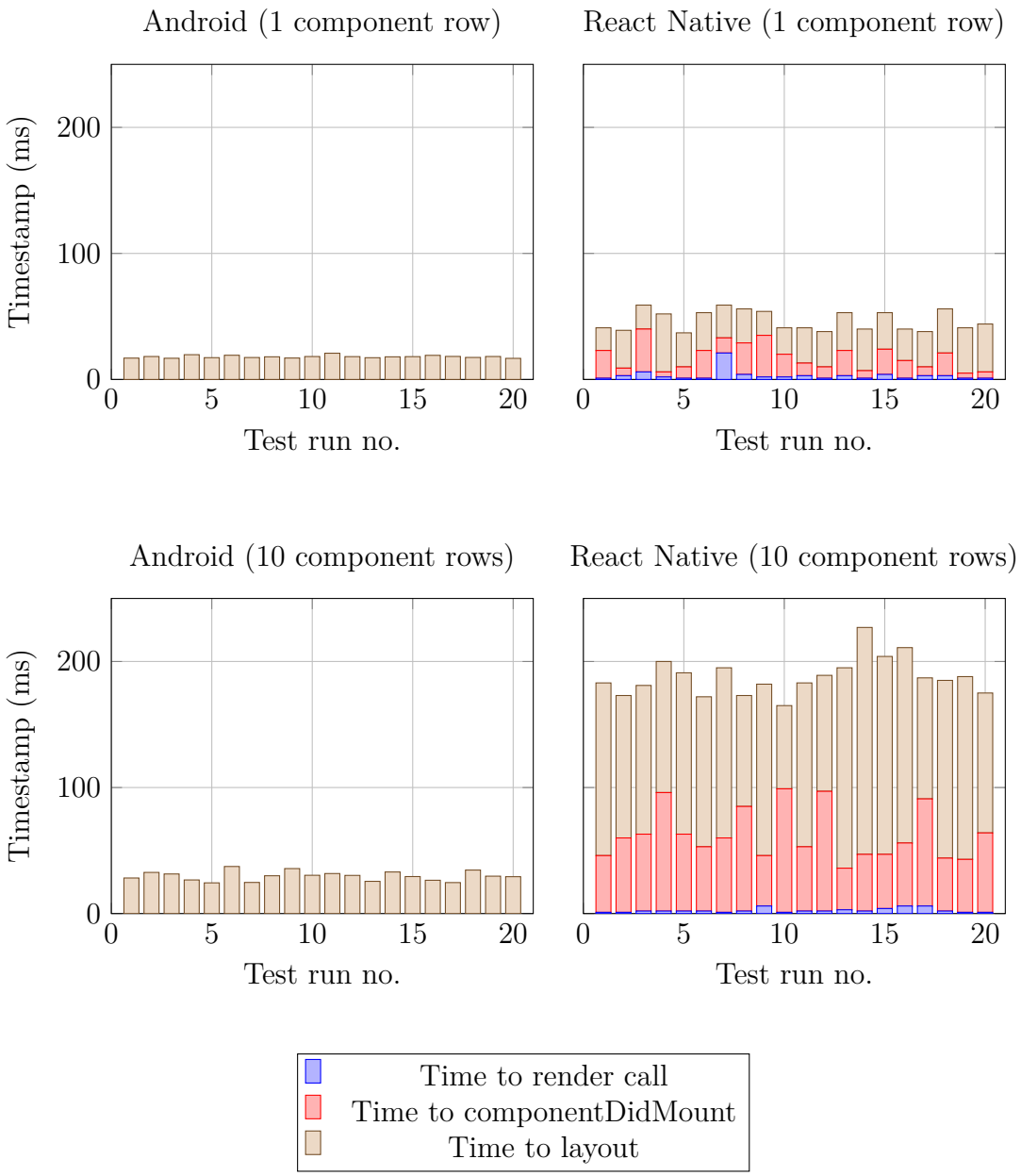


Figure 10: Tests with 1 and 10 rows of fundamental components

	Android (1 row)	RN (1 row)	Android (10 rows)	RN (10 rows)
$\overline{x}$	18.0	46.8	29.8	187.9
σ^2	1.0	8.0	3.7	14.7
min	16.7	37.0	24.4	165.0
max	20.8	59.0	37.4	227.0

Table 4: Total time to layout for 1/10 row(s) of components

100 rows of components: In this second test we will compare how React Native handles a view with lots of components compared to native Android. Our view will contain:

- 100 TextView components
- 100 Button components
- 100 ImageView components showing a local image

This will show how each platform handles an excessive amount of components during initial render, and whether there are any significant differences between the platforms here. This test will be performed on a 2018 Huawei P20 Pro phone in addition to the LG Nexus 5. This will show how latency compares between the platforms on newer devices. This test should put enough load on the newer phone to make results interesting, where the previous tests likely would have completed so fast as to be limited by VSync.

The results reveal a massive difference between the phones used, especially in the React Native case. In that particular test case, there is a four-fold decrease in latency on the newer device. The difference is not nearly as significant between the Native Android test configurations. This time around the results are consistent, which we speculate is due to the large amount of work that needs to be done. Small fluctuations affect the test results much less than before at these timescales.

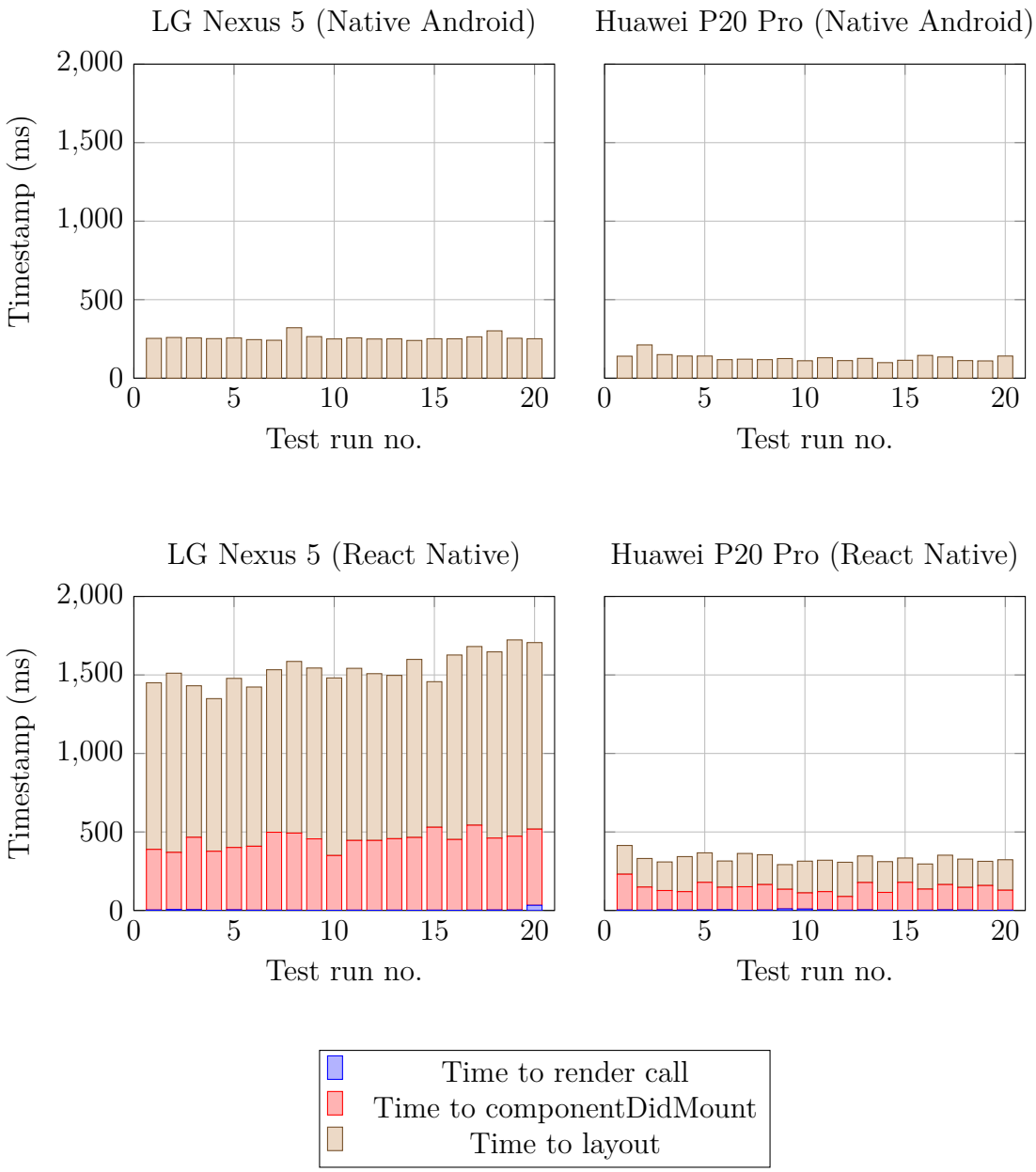


Figure 11: 100 rows of fundamental components

	Android (LG)	RN (LG)	Android (Huawei)	RN (Huawei)
$\overline{x}$	258.4	1538.7	130.1	331.6
σ^2	19.3	101.0	23.9	28.9
min	240	1349	99	292
max	321	1723	212	414

Table 5: Total time to layout for 100 rows of components

5.3.2 Navigation

In the navigation test we measure how long it takes for both an Android and a React Native application to navigate to another view. The measurement start point is a log entry of when the user initiated the navgiation action. The target view finishing it's rendering pass marks the end of the measurement.

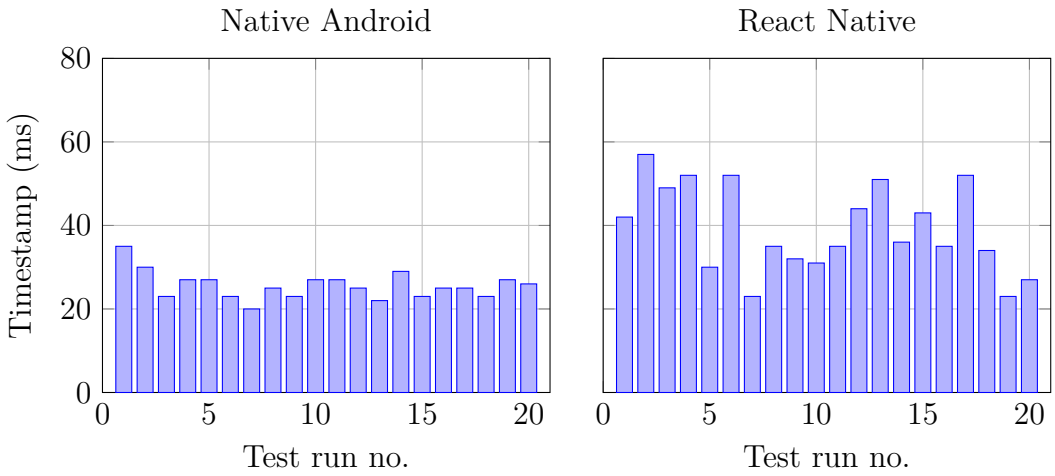


Figure 12: Navigation test measurements

	Native Android	React Native
$\overline{x}$	25.6	39.1
σ^2	3.3	10.4
min	20	23
max	35	57

Table 6: Navigation measurements

Figure 12 shows the results of 20 navigation test runs on the native Android and React Native apps respectively. We can see that the native Android navigation action time is very consistent with a mean of 25.6 and a standard deviation of only 3.3, while the React Native navigation action time has a mean of 39.1 with a standard deviation of 10.4 (see Table 6).

6 Discussion

6.1 Application launch

The application launch tests show that a React Native application does load significantly slower than a native Android application. We see a constant time to initial render penalty of up to three times slower with load times around 700 ms instead of around 220 ms. Unlike a native app which only needs to load the native runtime, a React Native app also has to start a JavaScript engine and run a JS application within it in order to know what to initially render, and this really shows in the results.

There are ways to alleviate a longer application launch. For instance, many popular mobile applications show a splash screen while performing initializations. This can give reassurance to the user that the application is working as intended, and still loading content. With a little bit of native code a React Native application can also show a splash screen instead of the blank screen that can be seen by default.

Keep in mind that the test application was a minimal hello world example, which only renders a basic text view on both platforms. It only measured the overhead of the React Native runtime, and the time to initial render. A real application would do this followed by other initialization tasks such as fetching content from the Internet or flash memory. Any subsequent operations are expected to take approximately the same amount of extra time on both platforms. As such, more complex applications will see a diminishing difference in launch times between native and React Native. More modern devices with better performance also improve the launch times, further reducing the impact React Native has in this category. In fact in this test, the Huawei P20 Pro React Native launch times almost matched the Native Android app launch times, with a mean launch time of 296.3 ms versus 178.3 ms.

Depending on the type of application, launch times may present a significant shortcoming in the performance of React Native. A performance penalty of half a second, possibly even worse on slower hardware or in applications with lots of dependencies, is not something to overlook. It is up to the application developers to decide if the extra launch time is acceptable for their particular application, and a reasonable performance penalty to pay for the advantages React Native brings.

Improving launch times is a hard problem that is certainly not alleviated by the addition of a JavaScript runtime. Applications may have several megabytes of JavaScript code to parse, which further impacts launch times. React Native contains an optional advanced feature to unbundle code, so that only the necessary code is parsed and ran at app launch time. This should help alleviate problems is huge applications with lots of views that are used more seldom.

In the case of JavaScript, launch times are not only a problem for React Native apps and things are constantly improving across the ecosystem. Node.js and Electron are enabling easy development of JavaScript programs on desktop computers, and here launch times are equally important. For example the Atom text editor is written using Electron and JavaScript, and its developers are constantly trying to improve launch time performance. Atom recently managed to make use of a V8 JavaScript

engine feature called snapshots. Snapshots allow running parts of the application beforehand, and storing a snapshot of the memory contents for later restoration [8]. While React Native does not use V8 currently, perhaps a similar feature will be implemented into the JavaScriptCore engine which React Native uses, thus making it possible to make use of a similar optimization in the future.

6.2 React Native components

Testing and comparing various React Native components revealed that it is important to choose the right tool for the job, some components were more suited to a certain use case than others.

6.2.1 List components

ScrollView For short lists or views where content might overflow the screen, `<ScrollView>` is a suitable component providing scrolling capabilities to the view. Its advantages are simple usage, and no pop-in artifacts during scrolling. However the disadvantages are that it loads and renders all of it's child components during initial render. `<ScrollViews>` containing hundreds or thousands of child components will freeze the app momentarily during the initial render phase, and the `<ScrollView>` component is thus not suited for use-cases where large amounts of child components are needed.

FlatList React Native's built-in list component making use of virtualization, `<FlatList>`, is a good pick when rendering huge lists with items of varying dimensions. The component loads quickly, and it was quick enough for most normal use-cases. However, if the list scrolls fast enough there can be visible pop-in artifacting which can be annoying to handle. `<FlatList>` allows for tweaking a few parameters, and it's base component `<VirtualizedList>` provides for lots of customization, but we were unable to find any combination of props that would fix the pop-in issues and thus decided to stick to the defaults.

Flipkart's RecyclerListView manages to deliver impressive performance by borrowing ideas from native Android's RecyclerView. We found this component to be the best pick for huge lists where list items are mostly of the same sizes to aid the recycling process. With slight tweaks, all pop-in problems were gone and initial load times remained short enough that they could not be noticed. The component is permissively licensed under Apache 2.0 and available for easy installation on npm.

6.3 Native Android vs React Native

6.3.1 Fundamental components

In this test case we rendered a handful of fundamental UI components. The 1 row results do not yet show any meaningful differences between how long React Native takes to do the render compared to native Android. Latencies remain below 50 ms

on average, so even if the Native Android app is faster, so fast that it is consistently limited by VSync, we are only talking about a few frames faster. An observant user might not consider the action instantaneous anymore as with the required 20 ms latency figures cited earlier, but most users shouldn't really be bothered by such a low latency.

The 10 row test case shows a larger difference between the platforms on the LG Nexus 5 phone. The Native Android figures have increased enough that they are no longer limited by VSync, but still they only occasionally go above a two frame rendering time (33.3 ms) with a mean value of 29.8 ms. This means the Android figures have only just gotten out of the zone where they are capped by VSync, and are now expected to increase linearly when we increase the number of components.

The React Native case is a different story, because they were never capped by a lower bound. Rendering 10 rows of fundamental components on React Native takes up to 227 ms, with a mean value of 187.9 ms. This is already at a point where users may start noticing a slight delay, but not yet nearly enough to cause an annoyance in this use case.

These results point to React Native being able to comfortably render a screen full of components even on the older LG Nexus 5 phone. It is certainly outperformed by native code, but again perhaps not by enough that it will matter. This is another case where the developers will have to weigh their options, those demanding the best performance should use native code, but developers valuing the concept of cross-platform development and tools may want to consider React Native.

In the excessive number of components test, with 100 lines of fundamental components totaling a number of 300 components, we roughly see the expected tenfold increase in rendering time as the number of components increased with the same rate. Note that this should be an unrealistic scenario caused by lazy development practices, a proper application should avoid rendering this many components at once and instead use e.g. list components with virtualization. It still serves to highlight that the rendering times roughly follow a linear relationship with respect to the number of components to be rendered on both platforms.

In the 100 lines test, React Native causes a very annoying latency spike of around 1.5 seconds when rendering the view on a LG Nexus 5 phone. This would make the application frustrating to use and/or unusable. Note that the native application remains usable with a mean latency of 258.4 ms. On the more recent Huawei P20 Pro phone, we see a huge decrease in latency on both platforms. Latency is almost halved in the native implementation, and the React Native application still renders quickly enough for users not to be annoyed, especially if rendering this view was an infrequent operation. Rendering took 331.6 ms on average.

6.3.2 Navigation

The React Native results are surprisingly low here considering the results of the previous rendering latency test. In some instances React Native even beats native Android in this simple stack navigation test. The native Android application performs the navigation action by navigating to another activity, which may explain why

React Native can be faster as it simply swaps out the rendered components within an activity without actually switching into another activity.

However it is important to note that the React Native results are much more inconsistent, with a standard deviation of 10.4, while the Android results are comparatively consistent with a standard deviation of only 3.3. The mean values of both platforms stay at reasonable values: 25.6 ms for Android, and 39.1 for React Native. These are both fast enough to be considered instantaneous for a navigation action, which in addition uses a 200-300 ms transition animation by default that should mask any small latencies introduced here.

Another thing to keep in mind is that this test only measures the navigation action itself by rendering an extremely simple view after navigating in both test cases. The view only contained one component which draws the background in a black color. When navigating to a more complex view, one should expect similar behavior as in the previous test (Section 6.3.1), since the new view and all of it's child components will have to be drawn as part of the navigation event. As such a nontrivial application is expected to perform considerably faster with native code compared to React Native, but regardless a navigation action should not be the culprit for performance problems.

6.4 Summary

Due to the performance penalties introduced by React Native compared to native code, optimizing your application becomes important. A long list which works just fine in a native application might cause slowdowns and latency in React Native, and choosing the right list implementation is key to avoiding performance problems in this use case.

Even if the developers of an application agree on the ideas behind cross-platform tools and are comfortable with writing their application in JavaScript, it is still important to consider whether a small performance hit can be taken in exchange for the advantages cross-platform tools may bring to the development process. The impact of the performance penalty is also greatly dependent on the type of application that is to be developed, simpler applications will generally have a smaller performance impact when using cross-platform tools compared to native code.

Another point to note when deciding which framework to use for an application is how many generations of old phones the application should support. We saw that on newer devices React Native performance was improved significantly and in many cases was close enough to the native equivalent as to not be of an issue. This could be a deciding factor, as many users do swap out their devices for newer ones at a rather fast pace.

While performance intensive applications can often be implemented in React Native by leveraging the possibility of calling native code from JavaScript, in the end it might not be worth it if most code ends up written in native code. As long as the UI is not overly complex though, React Native might be a good choice for the project. Resource intensive computations can be performed in native code, and then only the native part of the application has to be written individually for each

platform.

In the end it is up to the app developers to find a framework that they are comfortable with, and one that suits the particular project at hand. React Native is one possibility to consider, and this study provides some insight into what the possible impacts are of such a decision performance-wise.

7 Conclusions

In this thesis the performance of the React Native cross-platform framework is evaluated on Android through simple test applications. Performance is compared to native Android test applications to find out whether there is a meaningful performance impact introduced by React Native. The need for cross-platform frameworks is discussed, and the lack of previous in-depth academic study about React Native performance specifically is a motivation for the work.

A brief history of cross-platform frameworks and of React Native itself is presented in the background section. The concept and importance of latency is also discussed, together with an introduction to important concepts in React Native, followed by an analysis of existing related performance evaluations.

We introduce methods for measuring and comparing common performance aspects between React Native and native Android applications. Measurements were performed on application launch times, component render latency, navigation actions and list scrolling. Where possible, the implementations are such that results can be compared directly between React Native and Android test applications. This helps find out the exact amount of overhead caused by React Native compared to native Android code. In the list scrolling test we show how different React Native list components perform during heavy usage, and which component is best for a certain use-case.

Our results indicate that React Native does incur a significant performance impact especially on older hardware. On older devices such as the LG Nexus 5 we can see performance problems in application launch times, or when rendering large amounts of components. On the other hand, the results show that these issues are diminished on newer flagships such as the Huawei P20 Pro, where even an excessive amount of components can be rendered in time so as to not cause user frustration. The results will allow developers to make informed decisions when considering using the React Native framework for their next projects, and the presented methods will be valuable to studies performing further research on the subject.

References

[1] Abi management | android developers. https://developer.android.com/ndk/guides/abis.html#sa. (Accessed on 04/17/2018).

[2] Android graphics pipeline: From button to framebuffer (part 1). https://www.inovex.de/blog/android-graphics-pipeline-from-button-to-framebuffer-part-1/. (Accessed on 03/13/2018).

[3] Animations and performance | web fundamentals | google developers. https://developers.google.com/web/fundamentals/design-and-ux/animations/animations-and-performance. (Accessed on 05/18/2018).

[4] Architectural overview of cordova platform - apache cordova. https://cordova.apache.org/docs/en/latest/guide/overview/index.html#architecture. (Accessed on 04/24/2018).

[5] clock_gettime(3): clock/time functions - linux man page. https://linux.die.net/man/3/clock_gettime. (Accessed on 02/28/2018).

[6] How android draws views | android developers. https://developer.android.com/guide/topics/ui/how-android-draws.html. (Accessed on 04/13/2018).

[7] Idc: Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/os. (Accessed on 05/09/2018).

[8] Improving startup time | atom blog. https://blog.atom.io/2017/04/18/improving-startup-time.html. (Accessed on 05/18/2018).

[9] Launch-time performance | android developers. https://developer.android.com/topic/performance/launch-time. (Accessed on 05/03/2018).

[10] Mobile operating system market share worldwide | statcounter global stats. http://gs.statcounter.com/os-market-share/mobile/worldwide. (Accessed on 05/09/2018).

[11] Modulecounts. http://www.modulecounts.com/. (Accessed on 04/24/2018).

[12] Navigating between screens · react native. https://facebook.github.io/react-native/docs/navigation.html. (Accessed on 05/11/2018).

[13] performance.now() - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now. (Accessed on 02/28/2018).

[14] Programming languages definition | tiobe - the software quality company. https://www.tiobe.com/tiobe-index/programming-languages-definition/. (Accessed on 05/03/2018).

[15] Propertycross. http://propertycross.com/. (Accessed on 02/21/2018).

[16] React native performance case study, how it differs from native apps: Part 1 (messagequeue & js.... https://medium.com/@rotemmiz/react-native-internals-a-wider-picture-part-1-messagequeue-js-thread-7894a7cba868. (Accessed on 04/17/2018).

[17] React navigation (v2) · routing and navigation for your react native apps. https://reactnavigation.org/. (Accessed on 05/11/2018).

[18] Recyclerlistview: High performance listview for react native and web. https://medium.com/@naqvitalha/recyclerlistview-high-performance-listview-for-react-native-and-web-e368d6f0d7ef. (Accessed on 05/09/2018).

[19] Recycling rows for high performance react native list views. https://medium.com/@talkol/recycling-rows-for-high-performance-react-native-list-views-628fd0363861. (Accessed on 05/09/2018).

[20] Stack overflow developer survey 2018. https://insights.stackoverflow.com/survey/2018#overview. (Accessed on 04/24/2018).

[21] Tiobe index | tiobe - the software quality company. https://www.tiobe.com/tiobe-index/. (Accessed on 05/03/2018).

[22] Viewtreeobserver | android developers. https://developer.android.com/reference/android/view/ViewTreeObserver.html. (Accessed on 04/10/2018).

[23] Integration with existing apps · react native. https://facebook.github.io/react-native/docs/integration-with-existing-apps.html, 2018. (Accessed on 03/20/2018).

[24] Anderson, G., Doherty, R., and Ganapathy, S. User perception of touch screen latency. In *Design, User Experience, and Usability. Theory, Methods, Tools and Practice* (Berlin, Heidelberg, 2011), A. Marcus, Ed., Springer Berlin Heidelberg, pp. 195–202.

[25] Andersson, J. Using react native and aws lambda for cross-platform development in a startup. Master's thesis, Linköping University, Software and Systems, 2017.

[26] Apple Inc. Accessibility Traits | Apple Developer Documentation. https://developer.apple.com/documentation/uikit/accessibility/uiaccessibility/accessibility_traits. (Accessed on 02/05/2018).

[27] Axelsson, O., and Carlström, F. Evaluation targeting react native in comparison to native mobile development, 2016. Student Paper.

[28] Bosnic, S., Papp, I., and Novak, S. The development of hybrid mobile applications with apache cordova. In *2016 24th Telecommunications Forum (TELFOR)* (Nov 2016), pp. 1–4.

[29] Boushehrinejadmoradi, N., Ganapathy, V., Nagarakatte, S., and Iftode, L. Testing cross-platform mobile app development frameworks (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2015), pp. 441–451.

[30] Brahler, S. Analysis of the android architecture. *Karlsruhe institute for technology 7*, 8 (2010).

[31] Čarapina, M., Mekterović, I., Jagušt, T., Drljević, N., Baksa, J., Kovačević, P., and Botički, I. Developing a multiplatform solution for mobile learning. In *International Conference on Computers in Education (23; 2015)* (2015).

[32] Corral, L., Sillitti, A., and Succi, G. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science 10* (2012), 736–743.

[33] Facebook Inc. View, React Native 0.52 documentation. https://facebook.github.io/react-native/docs/0.52/view.html. (Accessed on 02/05/2018).

[34] Facebook Inc. Performance. https://facebook.github.io/react-native/docs/performance.html, 2018. [Online; accessed 18-January-2018].

[35] Gao, Y., Luo, Y., Chen, D., Huang, H., Dong, W., Xia, M., Liu, X., and Bu, J. Every pixel counts: Fine-grained ui rendering analysis for mobile applications. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications* (May 2017), pp. 1–9.

[36] Hansson, N., and Vidhall, T. Effects on performance and usability for cross-platform application development using react native. Master's thesis, Linköping University, Human-Centered systems, 2016.

[37] Jota, R., Ng, A., Dietz, P., and Wigdor, D. How fast is fast enough?: A study of the effects of latency in direct-touch pointing tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 2291–2300.

[38] Kämäräinen, T., Siekkinen, M., Ylä-Jääski, A., Zhang, W., and Hui, P. Dissecting the end-to-end latency of interactive mobile video applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2017), HotMobile '17, ACM, pp. 61–66.

[39] Ma, Y., Liu, X., Liu, Y., Liu, Y., and Huang, G. A tale of two fashions: An empirical study on the performance of native apps and web apps on android. *IEEE Transactions on Mobile Computing 17*, 5 (May 2018), 990–1003.

[40] Majchrzak, T., and Grønli, T.-M. Comprehensive analysis of innovative cross-platform app development frameworks. In *Proceedings of the 50th Hawaii International Conference on System Sciences* (2017).

[41] Malavolta, I. Beyond native apps: Web technologies to the rescue! (keynote). In *Proceedings of the 1st International Workshop on Mobile Development* (New York, NY, USA, 2016), Mobile! 2016, ACM, pp. 1–2.

[42] Miller, R. B. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (New York, NY, USA, 1968), AFIPS '68 (Fall, part I), ACM, pp. 267–277.

[43] Ng, A., Lepinski, J., Wigdor, D., Sanders, S., and Dietz, P. Designing for low-latency direct-touch input. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2012), UIST '12, ACM, pp. 453–464.

[44] Rajkumar, S. K., Hrishikesh, A. A., Vaibhav, V. G., and Omkar, S. J. Implementation of news app based on cordova cross-platform. In *2017 2nd International Conference for Convergence in Technology (I2CT)* (April 2017), pp. 60–62.

[45] Shneiderman, B. Response time and display rate in human performance with computers. *ACM Comput. Surv. 16*, 3 (Sept. 1984), 265–285.

[46] Spence, E. Windows Phone Is Dead, Long Live Microsoft's Smartphone Dream. https://www.forbes.com/sites/ewanspence/2017/07/12/microsoft-windows-phone-windows10-mobile-strategy/, 2017. Online, accessed 18-January-2018.

[47] Staff, C. React: Facebook's functional turn on writing javascript. *Communications of the ACM 59*, 12 (2016), 56–62.

[48] Sun, M., Wei, T., and Lui, J. C. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 331–342.

[49] Willocx, M., Vossaert, J., and Naessens, V. A quantitative assessment of performance in mobile app development tools. In *Mobile Services (MS), 2015 IEEE International Conference on* (2015), IEEE, pp. 454–461.

A App launch measurement automation script

```
APPLICATION=com.launch

ACTIVITY=$APPLICATION/$APPLICATION.MainActivity
ITERATIONS=20
LAUNCH_WAIT=2
KILL_WAIT=2

echo "--- killing possible existing instance of application ---"
adb shell am force-stop $APPLICATION

sleep 1

echo "--- clearing logcat ---"
adb logcat -c

echo "--- increasing logcat buffer size ---"
adb logcat -G 16M

sleep 1

echo "--- starting test ---"

for (( i=1; i<=$ITERATIONS; i++ ))
do
  echo "--- launching application ---"
  adb shell am start -n $ACTIVITY
  sleep $LAUNCH_WAIT

  echo "--- killing application ---"
  adb shell am force-stop $APPLICATION
  sleep $KILL_WAIT
done

echo "--- saving results ---"
adb logcat \*:S ReactNativeJS:V ActivityManager:I |
  grep "Displayed\|measurement" > results.txt
```